PYTHON

Master Data Science & IA

Institut Supérieur Informatique

Mail: sybrahima31@gmail.om
Link : https://github.com/syibrahima31

Sy ibrahima

Plan

1. *Les tuples*
2. *Table de Hash*
3. *Les dictionnaires*
4. *Les ensembles*
5. *Les Exceptions*

Les tuples

- Le tuple est une séquence, on peut donc appliquer les opérations comme le test d'appartenance avec in, accéder aux différents éléments avec un crochet, faire du slicing dessus.
- un tuple peut référencer des objets complètement hétérogènes. C'est donc très proche de la liste.

- le tuple est un objet immuable. Ça veut dire qu'une fois qu'on a créé le tuple, on ne peut plus le modifier.

La syntaxe des tuples

- On crée un tuple en utilisant simplement des parenthèses ouvrantes et fermantes .

- On peut créer un tuple avec un élément avec la syntaxe suivante : t = (4,).

- On peut créer un tuple de plusieurs éléments, qui contient des objets complètement hétérogènes

- Une caractéristique importante du tuple c'est que les parenthèses sont facultatives.

```
>>> t = (1,)
>>> t
(1, )

>>> t =(1,2,3,4)
>>> t
(1,2,3,4)

>>> t = 1,2,3,4
>>> t
(1,2,3,4)
```

Les tuples

- Le tuple **unpacking** fonctionne de la manière suivante : nous avons dans un tuple deux variables, a et **b**, et on dit que ces variables sont égales à une séquence qui doit avoir le même nombre d'éléments que l'on a dans notre **tuple**.

- En Python, il existe également la notion de **extended tuple unpacking**.

```
# tuple unpacking
>>> (a,b) = (2,3)
>>> a
2
>>> b
3
```

```
# extended tuple unpacking
>>> *x,y = [1,2,3,4]
>>> x
[1,2,3]
>>> y
4
```

Tables de hash

Jusqu'à maintenant nous avons couvert les types séquence avec notamment **listes**, les chaînes de caractères et les **tuples**. Dans cette leçon, nous allons parlé des **tables de hash**, une structure de données qui permet de répondre à certaines limitations des **types séquence**.

Les limitations des types séquences

Les types séquence ont été optimisés pour l'accès, **la modification** et **l'effacement** en fonction d'un numéro de séquence. Cependant ces types n'ont pas été optimisés pour le **test d'appartenance**.

- Un test d'appartenance linéaire et dépendant du nombre d'éléments :

```
>>> %timeit "x" in range(100)
>>> %timeit "x" in range(1_000)
>>> %timeit  "x" in range(100_000)
```

- Un test d'appartenance linéaire et dépendant du nombre d'éléments

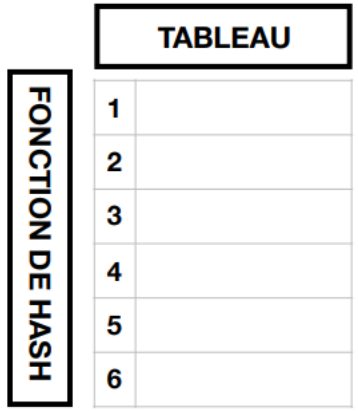
Mais supposons que nous avons des âges dans notre liste : t = [18, 35]. On pourrait vouloir écrire **t['alice'] = 35** pour lier un nom à un âge, donc indicer notre séquence non plus avec des entiers mais avec des chaînes de caractères. Et bien ça on ne peut pas le faire.

Tables de hash

La structure de données tables de hash permet de répondre à ces deux limitations

Une table de hash est constituée d'un tableau et d'une fonction dont le but est : quand on passe à notre fonction un objet, elle calcule une valeur comprise entre le nombre de lignes du tableau.

Ci-contre : Un tableau avec 6 éléments et une fonction de hash. Lorsqu'on lui passera un objet, la fonction calculera une valeur comprise entre 1 et 6. Le but de cette fonction est de créer une correspondance entre un objet quelconque et une case du tableau.
f(x) appartient à [1,6]
Cet ensemble **fonction de hash** et **tableau** constitue **une table de hash**.



Les dictionnaires

- Les dictionnaires sont des tables de hash. On a donc un temps **d'accès**, **d'insertion**, **d'effacement** et un **test d'appartenance** qui sont indépendants du nombre d'éléments.

- Les dictionnaires sont des objets **mutables**

- Dans un dictionnaire, on peut avoir comme clef n'importe quel objet qui est **hashable**, c'est-à dire un objet sur lequel on peut calculer la fonction de hash.

- En python tous les objets immuables sont hashables et que tous les objets mutables ne sont pas hashables

Création d'un dictionnaire

Pour initialiser un **dictionnaire** , on utile la syntaxe suivante:

```
>>> d = dict()
```

```
>>> d = {}
```

Les dictionnaires

Ajouter des valeurs dans un dictionnaire

Pour ajouter des valeurs à un **dictionnaire** il faut indiquer une clé ainsi qu'une valeur:

```
>>> a = {}
>>> a["nom"] = "Wayne"
>>> a["prenom"] = "Bruce"
>>> a
{'nom': 'Wayne', 'prenom': 'Bruce'
```

Récuperer une valeur dans un dictionnaire

La méthode *get* vous permet de **récupérer une valeur** dans un **dictionnaire** et si la clé est introuvable, vous pouvez donner une valeur à retourner par défaut:

```
>>> data = {"name": "Wayne", "age": 45}
>>> data.get("name")
'Wayne'
>>> data.get("adresse", "Adresse inconnue")
'Adresse inconnue'
```

Les dictionnaires

Existence d'une clé dans un dcitionnaire

Vous pouvez utiliser la méthode *has_key* pour vérifier la présence d'une clé que vous cherchez(a supprimer)

```
>>> a.has_key("nom")
True
```

Supprimer une entrée dans un dictionnaire

Il est possible de supprimer une entrée en indiquant sa clé, comme pour les listes:

```
>>> del a["nom"]
>>> a
{'prenom': 'Bruce'}
```

Utilisation d'un tuple

Une des forces de python est la combinaison **tuple/dictionnaire** qui fait des merveilles dans certains cas comme lors de l'utilisation de coordonnées.

```
>>> b = {}
>>> b[(32)]=12
>>> b[(45)]=13
>>> b
{(4, 5): 13, (3, 2): 12}
```

Les dictionnaires

Fusionner des dictionnaires

La méthode update permet de **fusionner deux dictionaires** .

```
>>> a = {'nom': 'Wayne'}
>>> b = {'prenom': 'bruce'}
>>> a.update(b)
>>> print(a)
{'nom': 'Wayne', 'prenom': 'Bruce'}
```

Supprimer une entrée dans un dictionnaire

Comme pour toute variable, vous ne pouvez pas **copier** un **dictionnaire** en faisant *dic1 = dic2* :

```
>>> d = {"k1":"Bruce", "k2":"Wayne"}
>>> e = d
>>> d["k1"] = "XXX"
>>> e
{'k2': 'Wayne', 'k1': 'XXX'}
```

Pour créer une **copie indépendante** vous pouvez utiliser la méthode **copy** :

```
>>> d = {"k1":"Bruce", "k2":"Wayne"}
>>> e = d.copy()
>>> d["k1"] = "XXX"
>>> e
{'k2': 'Wayne', 'k1': 'Bruce'}
```

Les dictionnaires

Les vues

- Les méthodes .keys(), .values() et .items() retournent un objet particulier qu'on appelle une vue.
- En Python, une vue est un objet sur lequel on peut itérer. On peut donc faire une boucle for sur cet objet. Et on peut également faire un test d'appartenance, donc faire par exemple in directement sur cette vue.
- La caractéristique principale des vues, c'est qu'elles sont mises à jour en même temps que le dictionnaire.

```
>>> liste = [("a",1), ("c", 2)]
>>> d = dict(liste)
>>> valeurs = d.values
dict_values([1, 2])
>>> d["a"] = 100
valeurs
```

Les dictionnaires

Parcourir les dictionnaires

Une manière classique de parcourir les dictionnaires est d'utiliser une boucle for en utilisant la notation de tuple unpacking :

```
>>> for k, v in d.items():
        print(k,v)
```

```
a 100
c 2
```

L'itérateur sur les dictionnaires sans spécification de vue fonctionne directement sur les clefs

```
>>> for k in d:
        print(k)
```

```
a
c
```

Les ensembles

- Les sets sont très proches des dictionnaires. Comme les dictionnaires, ils permettent de faire des tests d'appartenance, d'accéder / effacer / modifier des éléments indépendamment du nombre d'éléments.

- Les sets sont également des objets mutables. Mais à la différence des dictionnaires, les sets ne stockent qu'une clef. Il n'y a pas de valeur correspondante.

- La raison c'est que le set a été optimisé et créé pour des opérations spécifiques. Par exemple pour garder uniquement le nombre d'éléments uniques d'une séquence.

- Une autre opération pour laquelle le set est très utilisé : c'est pour faire des tests d'appartenance sur les éléments d'une séquence.

Création d'un set

- On peut créer un set vide avec la fonction built-in set() : s = set(). Ce qui nous donne un objet de type set qui est vide.

- On peut également créer un set avec des accolades : s = {1, 2, 3, 'a', True}. La différence entre les notations d'un set et d'un dictionnaire, c'est que dans le set il n'y a pas la notation deux points (:) qui sépare la clef et la valeur.

- On peut aussi créer un set à partir d'une séquence.

Les ensembles

Création d'un set

```
>>> set()
set()
>>> {1,2,3}
{1, 2, 3}
>>> set([1,2,2])
{1, 2}
```

Manipulation d'un set

- La fonction built-in **len()** pour obtenir le nombre d'éléments
- Le test d'appartenance est fait avec l'instruction **in**.
- Dans l'ensemble s, on peut ajouter des éléments avec la méthode **.add()**
- On peut aussi ajouter une séquence d'éléments avec la méthode **.update()**
- On peut également faire des opérations d'ensemble classiques sur un set. Pour s1 : {1, 2, 3} et s2 = {3, 4, 5
- une différence entre deux ensemble s1 - s2
- Union : s1 | s2
- Intersection : s1 & s2 renvoie {3}

Les exceptions

En y a au moins deux types d'erreurs à distinguer : les *erreurs de* ***syntaxe*** et les ***exceptions***.

Les erreurs de syntaxe

Les erreurs de syntaxe, qui sont des erreurs d'analyse du code, sont peut-être celles que vous rencontrez le plus souvent lorsque vous êtes encore en phase d'apprentissage de Python :

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                   ^
SyntaxError: invalid syntax
```

Exceptions

C’est quand une instruction ou une expression est syntaxiquement correcte, et le programme génère une **erreur** lors de son exécution. Les erreurs détectées durant l'exécution sont appelées des ***exceptions*** .

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

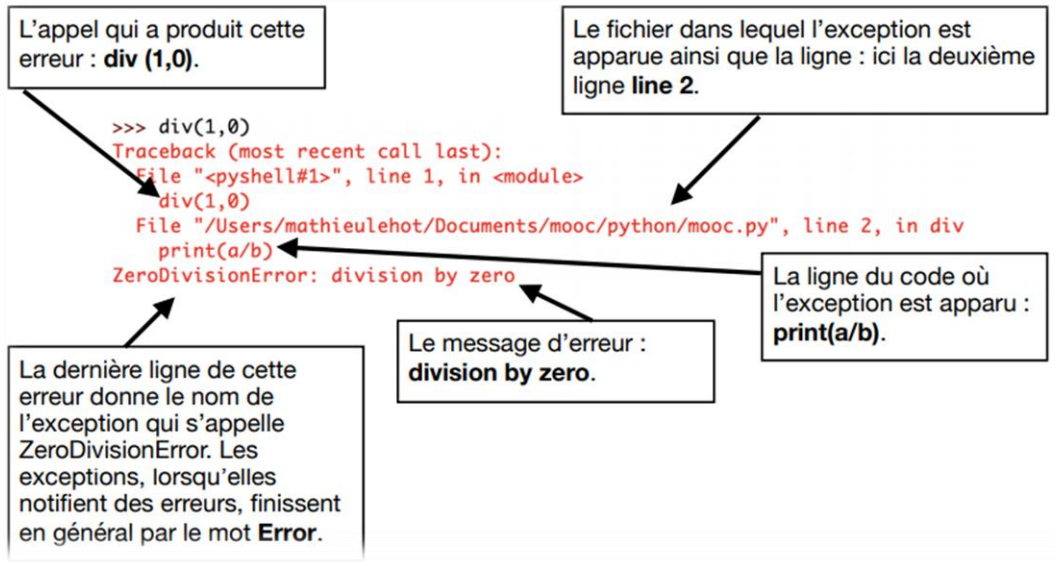
Les exceptions

Comprendre une exception

On commence a définir une fonction simple appelée une function simple : **div**

```
>>> def div(a,b):
        print(a/b)
```

Si on fait un **div(1, 0)** Python renvoie une erreur d'exécution, la division par 0 étant impossible.



Les exceptions

Capture d'une exception

- Pour capturer une exception on utilise un bloc **try… except**

- Tout ce qui est entre le **try** et le **except** va être évalué et si on a une exception qui se produit dans ce bloc de code, Python va regarder si cette exception a été capturée par le code.

```
>>> def div(a,b):
        try :
            print(a/b)
        except ZeroDivisionError:
            print("attention divion par zero non permit")
        print("continue")
```

Dans la suite si on exécute **div(1, 0),** l'exception est produite mais elle est correctement capturée par la clause **except**.

Les exceptions

Capture d'une exception

Si nous exécutons notre fonction **div** avec un chiffre en type chaîne de caractère : **div(1, '0')** . Python renvoie une erreur qui s'appelle **TypeError**.

```
>>> div(1,"0")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-55-e308882cfdda> in <module>
----> 1 div(1,"0")

<ipython-input-53-355f3312ce54> in div(a, b)
      1 def div(a,b):
      2     try :
----> 3         print(a/b)
      4     except ZeroDivisionError:
      5         print("attention divion par zero non permit")

TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

- Python donne la possibilité de capturer cette exception en , ajoutant une autre clause **except**
- On peut ajouter autant de clauses except qu'on le veut pour réagir à des exceptions particulières.

```
def div(a,b):
    try :
        print(a/b)
    except ZeroDivisionError:
        print("attention divion par zero non permit")
    except TypeError :
        print("On divise avec des entiers")
    print("continue")
```

Les exceptions

Mécanisme de bubbling

Une caractéristique importante des exceptions c'est qu'elles **bubble**. Ça veut dire qu'elle vont remonter la pile d'exécution jusqu'à arrêter le programme.

Prenons deux fonctions **div** et **f** :

```
>>> def div(a,b):
        print(a/b)

>>> def f(x):
        div(1, x)
```

Si maintenant on fait f(0)

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-61-6bfbdbfff9c4> in <module>
----> 1 f(0)

<ipython-input-59-79bd823ceb05> in f(x)
      1 def f(x):
----> 2     div(1, x)

<ipython-input-58-338f0144455e> in div(a, b)
      1 def div(a,b):
----> 2     print(a/b)

ZeroDivisionError: division by zero
```

Les exceptions

Mécanisme de bubbling

Ce mécanisme de bubbling a deux avantages majeurs :

- on peut capturer l'exception n'importe où le long de la pile d'exécution
- notre trace d'exécution est capable de dire exactement par où est passée l'exception et fournit donc des informations précieuses sur le diagnostic du problème dans notre programme